Statistically Rigorous Java Performance Evaluation

Andy Georges Dries Buytaert Lieven Eeckhout

Department of Electronics and Information Systems, Ghent University, Belgium

{ageorges,dbuytaer,leeckhou}@elis.ugent.be

Abstract

Java performance is far from being trivial to benchmark because it is affected by various factors such as the Java application, its input, the virtual machine, the garbage collector, the heap size, etc. In addition, non-determinism at run-time causes the execution time of a Java program to differ from run to run. There are a number of sources of non-determinism such as Just-In-Time (JIT) compilation and optimization in the virtual machine (VM) driven by timer-based method sampling, thread scheduling, garbage collection, and various system effects.

There exist a wide variety of Java performance evaluation methodologies used by researchers and benchmarkers. These methodologies differ from each other in a number of ways. Some report average performance over a number of runs of the same experiment; others report the best or second best performance observed; yet others report the worst. Some iterate the benchmark multiple times within a single VM invocation; others consider multiple VM invocations and iterate a single benchmark execution; yet others consider multiple VM invocations and iterate the benchmark multiple times.

This paper shows that prevalent methodologies can be misleading, and can even lead to incorrect conclusions. The reason is that the data analysis is not statistically rigorous. In this paper, we present a survey of existing Java performance evaluation methodologies and discuss the importance of statistically rigorous data analysis for dealing with non-determinism. We advocate approaches to quantify startup as well as steady-state performance, and, in addition, we provide the JavaStats software to automatically obtain performance numbers in a rigorous manner. Although this paper focuses on Java performance evaluation, many of the issues addressed in this paper also apply to other programming languages and systems that build on a managed runtime system.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Experimentation, Measurement, Performance

Keywords Java, benchmarking, data analysis, methodology, statistics

1. Introduction

Benchmarking is at the heart of experimental computer science research and development. Market analysts compare commercial products based on published performance numbers. Developers benchmark products under development to assess their performance. And researchers use benchmarking to evaluate the impact on performance of their novel research ideas. As such, it is absolutely crucial to have a rigorous benchmarking methodology. A non-rigorous methodology may skew the overall picture, and may even lead to incorrect conclusions. And this may drive research and development in a non-productive direction, or may lead to a non-optimal product brought to market.

Managed runtime systems are particularly challenging to benchmark because there are numerous factors affecting overall performance, which is of lesser concern when it comes to benchmarking compiled programming languages such as C. Benchmarkers are well aware of the difficulty in quantifying managed runtime system performance which is illustrated by a number of research papers published over the past few years showing the complex interactions between low-level events and overall performance [5, 11, 12, 17, 24]. More specifically, recent work on Java performance methodologies [7, 10] stressed the importance of a well chosen and well motivated *experimental design*: it was pointed out that the results presented in a Java performance study are subject to the benchmarks, the inputs, the VM, the heap size, and the hardware platform that are chosen in the experimental setup. Not appropriately considering and motivating one of these key aspects, or not appropriately describing the context within which the results were obtained and how they should be interpreted may give a skewed view, and may even be misleading or at worst be incorrect.

The orthogonal axis to experimental design in a performance evaluation methodology, is *data analysis*, or how to analyze and report the results. More specifically, a performance evaluation methodology needs to adequately deal

with the non-determinism in the experimental setup. In a Java system, or managed runtime system in general, there are a number of sources of non-determinism that affect overall performance. One potential source of non-determinism is Just-In-Time (JIT) compilation. A virtual machine (VM) that uses timer-based sampling to drive the VM compilation and optimization subsystem may lead to non-determinism and execution time variability: different executions of the same program may result in different samples being taken and, by consequence, different methods being compiled and optimized to different levels of optimization. Another source of non-determinism comes from thread scheduling in time-shared and multiprocessor systems. Running multithreaded workloads, as is the case for most Java programs, requires thread scheduling in the operating system and/or virtual machine. Different executions of the same program may introduce different thread schedules, and may result in different interactions between threads, affecting overall performance. The non-determinism introduced by JIT compilation and thread scheduling may affect the points in time where garbage collections occur. Garbage collection in its turn may affect program locality, and thus memory system performance as well as overall system performance. Yet another source of non-determinism is various system effects, such as system interrupts — this is not specific to managed runtime systems though as it is a general concern when running experiments on real hardware.

From an extensive literature survey, we found that there are a plethora of prevalent approaches, both in experimental design and data analysis for benchmarking Java performance. Prevalent data analysis approaches for dealing with non-determinism are not statistically rigorous though. Some report the average performance number across multiple runs of the same experiments; others report the best performance number, others report the second best performance number and yet others report the worst. In this paper, we argue that not appropriately specifying the experimental design and not using a statistically rigorous data analysis can be misleading and can even lead to incorrect conclusions. This paper advocates using statistics theory as a rigorous data analysis approach for dealing with the non-determinism in managed runtime systems.

The pitfall in using a prevalent method is illustrated in Figure 1 which compares the execution time for running Jikes RVM with five garbage collectors (CopyMS, GenCopy, GenMS, MarkSweep and SemiSpace) for the SPECjvm98 db benchmark with a 120MB heap size — the experimental setup will be detailed later. This graph compares the prevalent 'best' method which reports the best performance number (or smallest execution time) among 30 measurements against a statistically rigorous method which reports 95% confidence intervals; the 'best' method does not control non-determinism, and corresponds to the SPEC reporting rules [23]. Based on the best method, one would

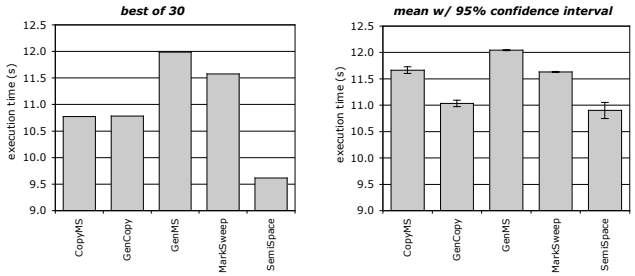

Figure 1. An example illustrating the pitfall of prevalent Java performance data analysis methods: the 'best' method is shown on the left and the statistically rigorous method is shown on the right. This is for db and a 120MB heap size.

conclude that the performance for the CopyMS and GenCopy collectors is about the same. The statistically rigorous method though shows that GenCopy significantly outperforms CopyMS. Similarly, based on the best method, one would conclude that SemiSpace clearly outperforms GenCopy. The reality though is that the confidence intervals for both garbage collectors overlap and, as a result, the performance difference seen between both garbage collectors is likely due to the random performance variations in the system under measurement. In fact, we observe a large performance variation for SemiSpace, and at least one really good run along with a large number of less impressive runs. The 'best' method reports the really good run whereas a statistically rigorous approach reliably reports that the average scores for GenCopy and SemiSpace are very close to each other.

This paper makes the following contributions:

- We demonstrate that there is a major pitfall associated with today's prevalent Java performance evaluation methodologies, especially in terms of data analysis. The pitfall is that they may yield misleading and even incorrect conclusions. The reason is that the data analysis employed by these methodologies is not statistically rigorous.
- We advocate adding statistical rigor to performance evaluation studies of managed runtime systems, and in particular Java systems. The motivation for statistically rigorous data analysis is that statistics, and in particular confidence intervals, enable one to determine whether differences observed in measurements are due to random fluctuations in the measurements or due to actual differences in the alternatives compared against each other. We discuss how to compute confidence intervals and discuss techniques to compare multiple alternatives.
- We survey existing performance evaluation methodologies for start-up and steady-state performance, and advocate the following methods. For start-up performance, we advise to: (i) take multiple measurements where each

measurement comprises one VM invocation and a single benchmark iteration, and (ii) compute confidence intervals across these measurements. For steady-state performance, we advise to: (i) take multiple measurements where each measurement comprises one VM invocation and multiple benchmark iterations, (ii) in each of these measurements, collect performance numbers for different iterations once performance reaches steady-state, i.e., after the start-up phase, and (iii) compute confidence intervals across these measurements (multiple benchmark iterations across multiple VM invocations).

- We provide publicly available software, called JavaStats, to enable a benchmarker to easily collect the information required to do a statistically rigorous Java performance analysis. In particular, JavaStats monitors the variability observed in the measurements to determine the number of measurements that need to be taken to reach a desired confidence interval for a given confidence level. JavaStats readily works for both the SPECjvm98 and DaCapo benchmark suites, and is available at `http://www.elis.ugent.be/JavaStats`.

This paper is organized as follows. We first present a survey in Section 2 on Java performance evaluation methodologies in use today. Subsequently, in Section 3, we discuss general statistics theory and how it applies to Java performance analysis. Section 4 then translates these theoretical concepts to practical methodologies for quantifying startup and steady-state performance. After detailing our experimental setup in Section 5, we then assess in Section 6 the prevalent evaluation methodologies compared to the statistically rigorous methodologies advocated in this paper. We show that in many practical situations, prevalent methodologies can be misleading, or even yield incorrect conclusions. Finally, we summarize and conclude in Section 7.

2. Prevalent Java Performance Evaluation Methodologies

There is a wide range of Java performance evaluation methodologies in use today. In order to illustrate this, we have performed a survey among the Java performance papers published in the last few years (from 2000 onwards) in premier conferences such as Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Programming Language Design and Implementation (PLDI), Virtual Execution Environments (VEE), Memory Management (ISMM) and Code Generation and Optimization (CGO). In total, we examined the methodology used in 50 papers.

Surprisingly enough, about one third of the papers (16 out of the 50 papers) does not specify the methodology used in the paper. This not only makes it difficult for other researchers to reproduce the results presented in the paper, it also makes understanding and interpreting the results hard.

During our survey, we found that not specifying or only partially specifying the performance evaluation methodology is particularly the case for not too recent papers, more specifically for papers between 2000 and 2003. More recent papers on the other hand, typically have a more detailed description of their methodology. This is in sync with the growing awareness of the importance of a rigorous performance evaluation methodology. For example, Eeckhout et al. [10] show that Java performance is dependent on the input set given to the Java application as well as on the virtual machine that runs the Java application. Blackburn et al. [7] confirm these findings and show that a Java performance evaluation methodology, next to considering multiple JVMs, should also consider multiple heap sizes as well as multiple hardware platforms. Choosing a particular heap size and/or a particular hardware platform may draw a fairly different picture and may even lead to opposite conclusions.

In spite of these recent advances towards a rigorous Java performance benchmarking methodology, there is no consensus among researchers on what methodology to use. In fact, almost all research groups come with their own methodology. We now discuss some general features of these prevalent methodologies and subsequently illustrate this using a number of example methodologies.

2.1 General methodology features

In the following discussion, we make a distinction between experimental design and data analysis. Experimental design refers to setting up the experiments to be run and requires good understanding of the system being measured. Data analysis refers to analyzing the data obtained from the experiments. As will become clear from this paper, both experimental design and data analysis are equally important in the overall performance evaluation methodology.

2.1.1 Data analysis

Average or median versus best versus worst run. Some methodologies report the average or median execution time across a number of runs — typically more than 3 runs are considered; some go up to 50 runs. Others report the best or second best performance number, and yet others report the worst performance number.

The SPEC run rules for example state that SPECjvm98 benchmarkers must run their Java application at least twice, and report both the best and worst of all runs. The intuition behind the worst performance number is to report a performance number that represents program execution intermingled with class loading and JIT compilation. The intuition behind the best performance number is to report a performance number where overall performance is mostly dominated by program execution, i.e., class loading and JIT compilation are less of a contributor to overall performance.

The most popular approaches are average and best — 8 and 10 papers out of the 50 papers in our survey, re-

spectively; median, second best and worst are less frequent, namely 4, 4 and 3 papers, respectively.

Confidence intervals versus a single performance number. In only a small minority of the research papers (4 out of 50), confidence intervals are reported to characterize the variability across multiple runs. The others papers though report a single performance number.

2.1.2 Experimental design

One VM invocation versus multiple VM invocations. The SPECjvm98 benchmark suite as well as the DaCapo benchmark suite come with a benchmark harness. The harness allows for running a Java benchmark multiple times within a single VM invocation. Throughout the paper, we will refer to multiple benchmark runs within a single VM invocation as benchmark *iterations*. In this scenario, the first iteration of the benchmark will perform class loading and most of the JIT (re)compilation; subsequent iterations will experience less (re)compilations. Researchers mostly interested in steady-state performance typically run their experiments in this scenario and report a performance number based on the subsequent iterations, not the first iteration. Researchers interested in startup performance will typically initiate multiple VM invocations running the benchmark only once.

Including compilation versus excluding compilation. Some researchers report performance numbers that include JIT compilation overhead, while others report performance numbers excluding JIT compilation overhead. In a managed runtime system, JIT (re)compilation is performed at run-time, and by consequence, becomes part of the overall execution. Some researchers want to exclude JIT compilation overhead from their performance numbers in order to isolate Java application performance and to make the measurements (more) deterministic, i.e., have less variability in the performance numbers across multiple runs.

A number of approaches have been proposed to exclude compilation overhead. One approach is to compile all methods executed during a first execution of the Java application, i.e., all methods executed are compiled to a predetermined optimization level, in some cases the highest optimization level. The second run, which is the timing run, does not do any compilation. Another approach, which is becoming increasingly popular, is called *replay compilation* [14, 20], which is used in 7 out of the 50 papers in our survey. In replay compilation, a number of runs are performed while logging the methods that are compiled and at which optimization level these methods are optimized. Based on this logging information, a compilation plan is determined. Some researchers select the methods that are optimized in the majority of the runs, and set the optimization level for the selected methods at the highest optimization levels observed in the majority of the runs; others pick the compilation plan that yields the best performance. Once the compilation plan is established, two benchmark runs are done in a single VM invocation: the first run does the compilation according to the compilation plan, the second run then is the timing run with adaptive (re)compilation turned off.

Forced GCs before measurement. Some researchers perform a full-heap garbage collection (GC) before doing a performance measurement. This reduces the non-determinism observed across multiple runs due to garbage collections kicking in at different times across different runs.

Other considerations. Other considerations concerning the experimental design include one hardware platform versus multiple hardware platforms; one heap size versus multiple heap sizes; a single VM implementation versus multiple VM implementations; and back-to-back measurements (‘aaabbb’) versus interleaved measurements (‘ababab’).

2.2 Example methodologies

To demonstrate the diversity in prevalent Java performance evaluation methodologies, both in terms of experimental design and data analysis, we refer to Table 1 which summarizes the main features of a number of example methodologies. We want to emphasize up front that our goal is not to pick on these researchers; we just want to illustrate the wide diversity in Java performance evaluation methodologies around today. In fact, this wide diversity illustrates the growing need for a rigorous performance evaluation methodology; many researchers struggle coming up with a methodology and, as a result, different research groups end up using different methodologies. The example methodologies summarized in Table 1 are among the most rigorous methodologies observed during our survey: these researchers clearly describe and/or motivate their methodology whereas many others do not.

For the sake of illustration, we now discuss three well described and well motivated Java performance methodologies in more detail.

Example 1. McGachey and Hosking [18] (methodology B in Table 1) iterate each benchmark 11 times within a single VM invocation. The first iteration compiles all methods at the highest optimization level. The subsequent 10 iterations do not include any compilation activity and are considered the timing iterations. Only the timing iterations are reported; the first compilation iteration is discarded. And a full-heap garbage collection is performed before each timing iteration. The performance number reported in the paper is the average performance over these 10 timing iterations along with a 90% confidence interval.

Example 2: Startup versus steady-state. Arnold et al. [1, 2] (methodologies F and G in Table 1) make a clear distinction between startup and steady-state performance. They evaluate the startup regime by timing the first run of a benchmark execution with a medium input set (`s10` for SPECjvm98). They report the minimum execution time

methodology	A	B	C	D	E	F	G	H	I	J	K	L	M
reference	[4]	[18]	[21]	[22]	[25]	[1]	[2]	[20]	[7, 14]	[23]	[8]	[3]	[9]
Data analysis													
average performance number from multiple runs	√	√	√									√	
median performance number from multiple runs							√						
best performance number from multiple runs					√	√	√		√	√	√		
second best performance number from multiple runs													√
worst performance number from multiple runs										√			
confidence interval from multiple runs		√		√									
Experimental design													
one VM invocation, one benchmark iteration												√	
one VM invocation, multiple benchmark iterations		√	√			√					√		
multiple VM invocations, one benchmark iteration					√	√							√
multiple VM invocations, multiple benchmark iterations				√			√						
including JIT compilation			√			√	√		√				
excluding JIT compilation		√			√			√	√				
all methods are compiled before measurement		√			√						√		
replay compilation								√	√				
full-heap garbage collection before measurement		√							√				
a single hardware platform	√	√	√		√	√	√	√		√	√		√
multiple hardware platforms				√					√			√	
a single heap size						√	√			√			
multiple heap sizes	√	√	√	√	√			√	√		√	√	√
a single VM implementation	√	√	√	√	√	√	√	√			√	√	√
multiple VM implementations									√	√			
back-to-back measurements													
interleaved measurements									√				

Table 1. Characterizing prevalent Java performance evaluation methodologies (in the columns) in terms of a number of features (in the rows): the '√' symbol means that the given methodology uses the given feature; the absence of the '√' symbol means the methodology does not use the given feature, or, at least, the feature is not documented.

across five benchmark executions, each benchmark execution triggering a new VM invocation. For measuring steady-state performance, in [1], Arnold et al. report the minimum execution time across five benchmark executions with a large input set (s100 for SPECjvm98) within a single VM invocation. Arnold et al. [2] use a different methodology for measuring steady-state performance. They do 10 experiments where each benchmark runs for approximately 4 minutes; this results in 10 times N runs. They then take the median execution time across these 10 experiments, resulting in N median execution times and then report the minimum median execution time. All the performance numbers reported include JIT compilation and optimization, as well as garbage collection activity.

Example 3: Replay compilation. Our third example methodology uses replay compilation to drive the performance evaluation. The idea of replay compilation discussed in [5, 7, 14, 20] (methodologies H and I in Table 1) is to build a compilation plan by running each benchmark n times with the adaptive runtime system enabled, logging the names of the methods that are optimized and their optimization levels. They then select the best compilation plan. The benchmarking experiment then proceeds as follows: (i) the first benchmark run performs compilation using the compilation plan, (ii) a full-heap garbage collection is performed, and (iii) the benchmark is run a second time with adaptive optimization turned off. This is done m times, and the best run is reported. Reporting the first benchmark run is called the mix method; reporting the second run is called the stable method.

3. Statistically Rigorous Performance Evaluation

We advocate statistically rigorous data analysis as an important part of a Java performance evaluation methodology. This section describes fundamental statistics theory as described in many statistics textbooks, see for example [15, 16, 19], and discusses how statistics theory applies to Java performance data analysis. The next section then discusses how to add statistical rigor in practice.

3.1 Errors in experimental measurements

As a first step, it is useful to classify errors as we observe them in experimental measurements in two main groups: *systematic errors* and *random errors*. Systematic errors are typically due to some experimental mistake or incorrect procedure which introduces a bias into the measurements. These errors obviously affect the accuracy of the results. It is up to the experimenter to control and eliminate systematic errors. If not, the overall conclusions, even with a statistically rigorous data analysis, may be misleading.

Random errors, on the other hand, are unpredictable and non-deterministic. They are unbiased in that a random error may decrease or increase a measurement. There may be many sources of random errors in the system. In practice, an important concern is the presence of perturbing events that are unrelated to what the experimenter is aiming at measuring, such as external system events, that cause outliers to appear in the measurements. Outliers need to be examined closely, and if the outliers are a result of a perturbing event, they should be discarded. Taking the best measurement also alleviates the issue with outliers, however, we advocate discarding outliers and applying statistically rigorous data analysis to the remaining measurements.

While it is impossible to predict random errors, it is possible to develop a statistical model to describe the overall effect of random errors on the experimental results, which we do next.

3.2 Confidence intervals for the mean

In each experiment, a number of samples is taken from an underlying population. A confidence interval for the mean derived from these samples then quantifies the range of values that have a given probability of including the actual population mean. While the way in which a confidence interval is computed is essentially similar for all experiments, a distinction needs to be made depending on the number of samples gathered from the underlying population [16]: (i) the number of samples n is large (typically, $n \geq 30$), and (ii) the number of samples n is small (typically, $n < 30$). We now discuss both cases.

3.2.1 When the number of measurements is large ($n \geq 30$)

Building a confidence interval requires that we have a number of measurements $x_i, 1 \leq i \leq n$, from a population with mean μ and variance σ^2. The mean of these measurements $\bar{x}$ is computed as

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}.$$

We will approximate the actual true value μ by the mean of our measurements $\bar{x}$ and we will compute a range of values $[c_1, c_2]$ around $\bar{x}$ that defines the confidence interval at a given probability (called the confidence level). The *confidence interval* $[c_1, c_2]$ is defined such that the probability of μ being between c_1 and c_2 equals $1 - \alpha$; α is called the *significance level* and $(1 - \alpha)$ is called the *confidence level*.

Computing the confidence interval builds on the central limit theory. The central limit theory states that, for large values of n (typically $n \geq 30$), $\bar{x}$ is approximately Gaussian distributed with mean μ and standard deviation $\sigma/\sqrt{n}$, provided that the samples $x_i, 1 \leq i \leq n$, are (i) independent and (ii) come from the same population with mean μ and finite standard deviation σ.

Because the significance level α is chosen a priori, we need to determine c_1 and c_2 such that $Pr[c_1 \leq \mu \leq c_2] = 1 - \alpha$ holds. Typically, c_1 and c_2 are chosen to form a symmetric interval around $\bar{x}$, i.e., $Pr[\mu < c_1] = Pr[\mu > c_2] = \alpha/2$. Applying the central-limit theorem, we find that

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}}$$

$$c_2 = \bar{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}},$$

with $\bar{x}$ the sample mean, n the number of measurements and s the sample standard deviation computed as follows:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}.$$

The value $z_{1-\alpha/2}$ is defined such that a random variable Z that is Gaussian distributed with mean $\mu = 0$ and variance $\sigma^2 = 1$, obeys the following property:

$$Pr[Z \leq z_{1-\alpha/2}] = 1 - \alpha/2.$$

The value $z_{1-\alpha/2}$ is typically obtained from a precomputed table.

3.2.2 When the number of measurements is small ($n < 30$)

A basic assumption made in the above derivation is that the sample variance s^2 provides a good estimate of the actual variance σ^2. This assumption enabled us to approximate $z = (\bar{x} - \mu)/(\sigma/\sqrt{n})$ as a standard normally distributed random variable, and by consequence to compute the confidence interval for $\bar{x}$. This is generally the case for experiments with a large number of samples, e.g., $n \geq 30$.

However, for a relatively small number of samples, which is typically assumed to mean $n < 30$, the sample variance s^2 can be significantly different from the actual variance σ^2 of the underlying population [16]. In this case, it can be shown that the distribution of the transformed value $t = (\bar{x} - \mu)/(s/\sqrt{n})$ follows the *Student's* t-distribution with $n - 1$ degrees of freedom. By consequence, the confidence interval can then be computed as:

$$c_1 = \bar{x} - t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}}$$

$$c_2 = \bar{x} + t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}},$$

with the value $t_{1-\alpha/2;n-1}$ defined such that a random variable T that follows the Student's t distribution with $n - 1$ degrees of freedom, obeys:

$$Pr[T \leq t_{1-\alpha/2;n-1}] = 1 - \alpha/2.$$

The value $t_{1-\alpha/2;n-1}$ is typically obtained from a precomputed table. It is interesting to note that as the number of measurements n increases, the Student t-distribution approaches the Gaussian distribution.

3.2.3 Discussion

Interpretation. In order to interpret experimental results with confidence intervals, we need to have a good understanding of what a confidence interval actually means. A 90% confidence interval, i.e., a confidence interval with a 90% confidence level, means that there is a 90% probability that the actual distribution mean of the underlying population, μ, is within the confidence interval. Increasing the confidence level to 95% means that we are increasing the probability that the actual mean is within the confidence interval. Since we do not change our measurements, the only way to increase the probability of the mean being within this new confidence interval is to increase its size. By consequence, a 95% confidence interval will be larger than a 90% confidence interval; likewise, a 99% confidence interval will be larger than a 95% confidence interval.

Note on normality. It is also important to emphasize that computing confidence intervals does not require that the underlying data is Gaussian or normally distributed. The central limit theory, which is at the foundation of the confidence interval computation, states that $\bar{x}$ is normally distributed irrespective of the underlying distribution of the population from which the measurements are taken. In other words, even if the population is not normally distributed, the average measurement mean $\bar{x}$ is approximately Gaussian distributed if the measurements are taken independently from each other.

3.3 Comparing two alternatives

So far, we were only concerned about computing the confidence interval for the mean of a single system. In terms of a Java performance evaluation setup, this is a single Java benchmark with a given input running on a single virtual machine with a given heap size running on a given hardware platform. However, in many practical situations, a researcher or benchmarker wants to compare the performance of two or more systems. In this section, we focus on comparing two alternatives; the next section then discusses comparing more than two alternatives. A practical use case scenario could be to compare the performance of two virtual machines running the same benchmark with a given heap size on a given hardware platform. Another example use case is comparing the performance of two garbage collectors for a given benchmark, heap size and virtual machine on a given hardware platform.

The simplest approach to comparing two alternatives is to determine whether the confidence intervals for the two sets of measurements overlap. If they do overlap, then we cannot conclude that the differences seen in the mean values are not due to random fluctuations in the measurements. In other words, the difference seen in the mean values is possibly due to random effects. If the confidence intervals do not overlap, however, we conclude that there is no evidence to suggest that there is not a statistically significant difference. Note the careful wording here. There is still a probability α that the differences observed in our measurements are simply due to random effects in our measurements. In other words, we cannot assure with a 100% certainty that there is an actual difference between the compared alternatives. In some cases, taking such 'weak' conclusions may not be very satisfying — people tend to like strong and affirmative conclusions — but it is the best we can do given the statistical nature of the measurements.

Consider now two alternatives with n_1 measurements of the first alternative and n_2 measurements of the second alternative. We then first determine the sample means $\bar{x}_1$ and $\bar{x}_2$ and the sample standard deviations s_1 and s_2. We subsequently compute the difference of the means as $\bar{x} = \bar{x}_1 - \bar{x}_2$. The standard deviation s_x of this difference of the mean values is then computed as:

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}.$$

The confidence interval for the difference of the means is then given by

$$c_1 = \bar{x} - z_{1-\alpha/2} s_x$$

$$c_2 = \bar{x} + z_{1-\alpha/2} s_x.$$

If this confidence interval includes zero, we can conclude, at the confidence level chosen, that there is no statistically significant difference between the two alternatives.

The above only holds in case the number of measurements is large on both systems, i.e., $n_1 \geq 30$ and $n_2 \geq 30$. In case the number of measurements on at least one of the two systems is smaller than 30, then we can no longer assume that the difference of the means is normally distributed. We then need to resort to the Student's t distribution by replacing the value $z_{1-\alpha/2}$ in the above formula with $t_{1-\alpha/2;n_{df}}$; the degrees of freedom n_{df} is then to be approximated by the integer number nearest to

$$n_{df} = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{(s_1^2/n_1)^2}{n_1 - 1} + \frac{(s_2^2/n_2)^2}{n_2 - 1}}.$$

3.4 Comparing more than two alternatives: ANOVA

The approach discussed in the previous section to comparing two alternatives is simple and intuitively appealing, however, it is limited to comparing two alternatives. A more general and more robust technique is called *Analysis of Variance (ANOVA)*. ANOVA separates the total variation in a set of measurements into a component due to random fluctuations in the measurements and a component due to the actual differences among the alternatives. In other words, ANOVA separates the total variation observed in (i) the variation observed *within* each alternative, which is assumed to be a result of random effects in the measurements, and (ii) the variation *between* the alternatives. If the variation between

the alternatives is larger than the variation within each alternative, then it can be concluded that there is a statistically significant difference between the alternatives. ANOVA assumes that the variance in measurement error is the same for all of the alternatives. Also, ANOVA assumes that the errors in the measurements for the different alternatives are independent and Gaussian distributed. However, ANOVA is fairly robust towards non-normality, especially in case there is a balanced number of measurements for each of the alternatives.

To present the general idea behind ANOVA it is convenient to organize the measurements as shown in Table 2: there are $n \cdot k$ measurements — n measurements for all k alternatives. The column means are defined as:

$$\bar{y}_{.j} = \frac{\sum_{i=1}^{n} y_{ij}}{n},$$

and the overall mean is defined as:

$$\bar{y}_{..} = \frac{\sum_{j=1}^{k}\sum_{i=1}^{n} y_{ij}}{n \cdot k}.$$

It is then useful to compute the variation due to the effects of the alternatives, sum-of-squares due to the alternatives (SSA), as the sum of the squares of the differences between the mean of the measurements for each alternative and the overall mean, or:

$$SSA = n\sum_{j=1}^{k}(\bar{y}_{.j} - \bar{y}_{..})^2.$$

The variation due to random effects within an alternative is computed as the sum of the squares of the differences (or errors) between the individual measurements and their respective alternative mean, or:

$$SSE = \sum_{j=1}^{k}\sum_{i=1}^{n}(y_{ij} - \bar{y}_{.j})^2.$$

Finally, the sum-of-squares total, SST, or the sum of squares of the differences between the individual measurements and the overall mean is defined as:

$$SST = \sum_{j=1}^{k}\sum_{i=1}^{n}(y_{ij} - \bar{y}_{..})^2.$$

It can be shown that

$$SST = SSA + SSE.$$

Or, in other words, the total variation observed can be split up into a *within* alternative (SSE) component and a *between* alternatives (SSA) component.

The intuitive understanding of an ANOVA analysis now is to quantify whether the variation across alternatives, SSA, is 'larger' in some statistical sense than the variation within each alternative, SSE, which is due to random measurement errors. A simple way of doing this is to compare the fractions SSA/SST versus SSE/SST. A statistically more rigorous approach is to apply a statistical test, called the *F-test*, which is used to test whether two variances are significantly different. We will not go into further detail here about how to apply the F-test, however, we refer the interested reader to a reference textbook on statistics, such as [16].

	Alternatives						
Measurements	1	2	...	j	...	k	Overall mean
1	y_{11}	y_{12}	...	y_{1j}	...	y_{1k}	
2	y_{21}	y_{22}	...	y_{2j}	...	y_{2k}	
⋮	⋮	⋮	⋱	⋮	⋱	⋮	
i	y_{i1}	y_{i2}	...	y_{ij}	...	y_{ik}	
⋮	⋮	⋮	⋱	⋮	⋱	⋮	
n	y_{n1}	y_{n2}	...	y_{nj}	...	y_{nk}	
Column means	$\bar{y}_{.1}$	$\bar{y}_{.2}$	...	$\bar{y}_{.j}$	...	$\bar{y}_{.k}$	$\bar{y}_{..}$

Table 2. Organizing the n measurements for k alternatives in an ANOVA analysis.

After completing an ANOVA test, we may conclude that there is a statistically significant difference between the alternatives, however, the ANOVA test does not tell us between which alternatives there is a statistically significant difference. There exists a number of techniques to find out between which alternatives there is or there is not a statistically significant difference. One approach, which we will be using in this paper, is called the Tukey HSD (Honestly Significantly Different) test. The advantage of the Tukey HSD test over simpler approaches, such as pairwise t-tests for comparing means, is that it limits the probability of making an incorrect conclusion in case there is no statistically significant difference between all the means and in case most of the means are equal but one or two are different. For a more detailed discussion, we refer to the specialized literature.

In summary, an ANOVA analysis allows for varying one *input variable* within the experiment. For example, in case a benchmarker wants to compare the performance of four virtual machines for a given benchmark, a given heap size and a given hardware platform, the virtual machine then is the input variable and the four virtual machines are the four alternatives. Another example where an ANOVA analysis can be used is when a benchmarker wants to compare the performance of various garbage collectors for a given virtual machine, a given benchmark and a given system setup.

3.5 Multi-factor and multivariate experiments

Multi-factor ANOVA. The ANOVA analysis discussed in the previous section is a so called one-factor ANOVA, meaning that only a single input variable can be varied during the setup. A multi-factor ANOVA allows for studying the effect of multiple input variables and all of their interactions, along with an indication of the magnitude of the measurement error. For example, an experiment where both the

garbage collector and the heap size are varied, could provide deep insight into the effect on overall performance of both the garbage collector and the heap size individually as well as the interaction of both the garbage collector and the heap size.

Multivariate ANOVA. The ANOVA analyses discussed so far only consider, what is called, a single dependent variable. In a Java context, this means that an ANOVA analysis only allows for making conclusions about a single benchmark. However, a benchmarker typically considers a number of benchmarks and is interested in the performance for all the benchmarks — it is important for a performance evaluation study to consider a large enough set of representative benchmarks. A multivariate ANOVA (MANOVA) allows for considering multiple dependent variables, or multiple benchmarks, within one single experiment. The key point of performing a MANOVA instead of multiple ANOVA analyses on the individual dependent variables, is that a MANOVA analysis takes into account the correlation across the dependent variables whereas multiple ANOVAs do not.

3.6 Discussion

In the previous sections, we explored a wide range of statistical techniques and we discussed how to apply these techniques within a Java performance evaluation context. However, using the more complex analyses, such as multi-factor ANOVA and MANOVA, raises two concerns. First, their output is often non-intuitive and in many cases hard to understand without deep background knowledge in statistics. Second, as mentioned before, doing all the measurements required as input to the analyses can be very time-consuming, up to the point where it becomes intractable. For these reasons, we limit ourselves to a Java performance evaluation methodology that is practical yet statistically rigorous. The methodology that we present computes confidence intervals which allows for doing comparisons between alternatives on a per-benchmark basis, as discussed in sections 3.3 and 3.4. Of course, a benchmarker who is knowledgeable in statistics may perform more complex analyses.

4. A practical statistically rigorous methodology

Having discussed the general theory of statistics and how it relates to Java performance evaluation, we suggest more practical and statistically rigorous methodologies for quantifying startup and steady-state Java performance by combining a number of existing approaches. The evaluation section in this paper then compares the accuracy of prevalent data analysis methodologies against these statistically rigorous methodologies.

Notation. We refer to x_{ij} as the measurement of the j-th benchmark iteration of the i-th VM invocation.

4.1 Startup performance

The goal of measuring start-up performance is to measure how quickly a Java virtual machine can execute a relatively short-running Java program. There are two key differences between startup and steady-state performance. First, startup performance includes class loading whereas steady-state performance does not, and, second, startup performance is affected by JIT compilation, substantially more than steady-state performance.

For measuring startup performance, we advocate a two-step methodology:

1. Measure the execution time of multiple VM invocations, each VM invocation running a single benchmark iteration. This results in p measurements x_{ij} with $1 \leq i \leq p$ and $j = 1$.
2. Compute the confidence interval for a given confidence level as described in Section 3.2. If there are more than 30 measurements, use the standard normal z-statistic; otherwise use the Student t-statistic.

Recall that the central limit theory assumes that the measurements are independent. This may not be true in practice, because the first VM invocation in a series of measurements may change system state that persists past this first VM invocation, such as dynamically loaded libraries persisting in physical memory or data persisting in the disk cache. To reach independence, we discard the first VM invocation for each benchmark from our measurements and only retain the subsequent measurements, as done by several other researchers; this assumes that the libraries are loaded when doing the measurements.

4.2 Steady-state performance

Steady-state performance concerns long-running applications for which start-up is of less interest, i.e., the application's total running time largely exceeds start-up time. Since most of the JIT compilation is performed during start-up, steady-state performance suffers less from variability due to JIT compilation. However, the other sources of non-determinism, such as thread scheduling and system effects, still remain under steady-state, and thus need to be considered.

There are two issues with quantifying steady-state performance. The first issue is to determine when steady-state performance is reached. Long-running applications typically run on large or streaming input data sets. Benchmarkers typically approximate long-running benchmarks by running existing benchmarks with short inputs multiple times within a single VM invocation, i.e., the benchmark is iterated multiple times. The question then is how many benchmark iterations do we need to consider before we reach steady-state performance within a single VM invocation? This is a difficult question to answer in general; the answer will differ

from application to application, and in some cases it may take a very long time before steady-state is reached.

The second issue with steady-state performance is that different VM invocations running multiple benchmark iterations may result in different steady-state performances [2]. Different methods may be optimized at different levels of optimization across different VM invocations, changing steady-state performance.

To address these two issues, we advocate a four-step methodology for quantifying steady-state performance:

1. Consider p VM invocations, each VM invocation running at most q benchmark iterations. Suppose that we want to retain k measurements per invocation.
2. For each VM invocation i, determine the iteration s_i where steady-state performance is reached, i.e., once the coefficient of variation (CoV)[1] of the k iterations ($s_i - k$ to s_i) falls below a preset threshold, say 0.01 or 0.02.
3. For each VM invocation, compute the mean $\bar{x}_i$ of the k benchmark iterations under steady-state:

$$\bar{x}_i = \sum_{j=s_i-k}^{s_i} x_{ij}.$$

4. Compute the confidence interval for a given confidence level across the computed means from the different VM invocations. The overall mean equals $\bar{x} = \sum_{i=1}^{p} \bar{x}_i$, and the confidence interval is computed over the $\bar{x}_i$ measurements.

We thus first compute the mean $\bar{x}_i$ across multiple iterations within a single VM invocation i, and subsequently compute the confidence interval across the p VM invocations using the $\bar{x}_i$ means, see steps 3 and 4 from above. The reason for doing so is to reach independence across the measurements from which we compute the confidence interval: the various iterations within a single VM invocation are not independent, however, the mean values $\bar{x}_i$ across multiple VM invocations are independent.

4.3 In practice

To facilitate the application of these start-up and steady-state performance evaluation methodologies, we provide publicly available software called JavaStats [2] that readily works with the SPECjvm98 and DaCapo benchmark suites. For startup performance, a script (i) triggers multiple VM invocations running a single benchmark iteration, (ii) monitors the execution time of each invocation, and (iii) computes the confidence interval for a given confidence level. If the confidence interval achieves a desired level of precision, i.e., the confidence interval is within 1% or 2% of the sample mean, the script stops the experiment, and reports the sample mean and its confidence interval. Or, if the desired level of precision is not reached after a preset number of runs, e.g., 30 runs, the obtained confidence interval is simply reported along with the sample mean.

For steady-state performance, JavaStats collects execution times across multiple VM invocations and across multiple benchmark iterations within a single VM invocation. JavaStats consists of a script running multiple VM invocations as well as a benchmark harness triggering multiple iterations within a single VM invocation. The output for steady-state performance is similar to what is reported above for startup performance.

SPECjvm98 as well as the DaCapo benchmark suite already come with a harness to set the desired number of benchmark iterations within a single VM invocation. The current version of the DaCapo harness also determines how many iterations are needed to achieve a desired level of coefficient of variation (CoV). As soon as the observed CoV drops below a given threshold (the convergence target) for a given window of iterations, the execution time for the next iteration is reported. JavaStats extends the existing harnesses (i) by enabling measurements across multiple VM invocations instead of a single VM invocation, and (ii) by computing and reporting confidence intervals.

These Java performance analysis methodologies do not control non-determinism. However, a statistically rigorous data analysis approach can also be applied together with an experimental design that controls the non-determinism, such as replay compilation. Confidence intervals can be used to quantify the remaining random fluctuations in the system under measurement.

A final note that we would like to make is that collecting the measurements for a statistically rigorous data analysis can be time-consuming, especially if the experiment needs a large number of VM invocations and multiple benchmark iterations per VM invocation (in case of steady-state performance). Under time pressure, statistically rigorous data analysis can still be applied considering a limited number of measurements, however, the confidence intervals will be looser.

5. Experimental Setup

The next section will evaluate prevalent data analysis approaches against statistically rigorous data analysis. For doing so, we consider an experiment in which we compare various garbage collection (GC) strategies — similar to what is being done in the GC research literature. This section discusses the experimental setup: the virtual machine configurations, the benchmarks and the hardware platforms.

5.1 Virtual machine and GC strategies

We use the Jikes Research Virtual Machine (RVM) [1] which is an open source Java virtual machine written in Java. Jikes RVM employs baseline compilation to compile a method

[1] CoV is defined as the standard deviation s divided by the mean $\bar{x}$.

[2] Available at `http://www.elis.UGent.be/JavaStats/`.

upon its first execution; hot methods are sampled by an OS-triggered sampling mechanism and subsequently scheduled for further optimization. There are three optimization levels in Jikes RVM: 0, 1 and 2. We use the February 12, 2007 SVN version of Jikes RVM in all of our experiments.

We consider five garbage collection strategies in total, all implemented in the Memory Management Toolkit (MMTk) [6], the garbage collection toolkit provided with the Jikes RVM. The five garbage collection strategies are: (i) CopyMS, (ii) GenCopy, (iii) GenMS, (iv) MarkSweep, and (v) SemiSpace; the generational collectors use a variable-size nursery. GC poses a complex space-time trade-off, and it is unclear which GC strategy is the winner without doing a detailed experimentation. We did not include the GenRC, MarkCompact and RefCount collectors from MMTk, because we were unable to successfully run Jikes with the GenRC and MarkCompact collector for some of the benchmarks; and RefCount did yield performance numbers that are statistically significantly worse than any other GC strategy across all benchmarks.

5.2 Benchmarks

Table 3 shows the benchmarks used in this study. We use the SPECjvm98 benchmarks [23] (first seven rows), as well as seven DaCapo benchmarks [7] (next seven rows). SPECjvm98 is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). The DaCapo benchmark is a recently introduced open-source benchmark suite; we use release version 2006-10-MR2. We use the seven benchmarks that execute properly on the February 12, 2007 SVN version of Jikes RVM. We use the default (medium size) input set for the DaCapo benchmarks.

In all of our experiments, we consider a per-benchmark heap size range, following [5]. We vary the heap size from a minimum heap size up to 6 times this minimum heap size, in steps of 0.25 times the minimum heap size. The per-benchmark minimum heap sizes are shown in Table 3.

5.3 Hardware platforms

Following the advice by Blackburn et al. [7], we consider multiple hardware platforms in our performance evaluation methodology: a 2.1GHz AMD Athlon XP, a 2.8GHz Intel Pentium 4, and a 1.42GHz Mac PowerPC G4 machine. The AMD Athlon and Intel Pentium 4 have 2GB of main memory; the Mac PowerPC G4 has 1GB of main memory. These machines run the Linux operating system, version 2.6.18. In all of our experiments we consider an otherwise idle and unloaded machine.

6. Evaluation

We now evaluate the proposed statistically rigorous Java performance data analysis methodology in three steps. We first measure Java program run-time variability. In a second step, we compare prevalent methods from Section 2 against the statistically rigorous method from Section 3. And as a final step, we demonstrate the use of the software provided to perform a statistically rigorous performance evaluation.

benchmark	description	min heap size (MB)
compress	file compression	24
jess	puzzle solving	32
db	database	32
javac	Java compiler	32
mpegaudio	MPEG decompression	16
mtrt	raytracing	32
jack	parsing	24
antlr	parsing	32
bloat	Java bytecode optimization	56
fop	PDF generation from XSL-FO	56
hsqldb	database	176
jython	Python interpreter	72
luindex	document indexing	32
pmd	Java class analysis	64

Table 3. SPECjvm98 (top seven) and DaCapo (bottom seven) benchmarks considered in this paper. The rightmost column indicates the minimum heap size, as a multiple of 8MB, for which all GC strategies run to completion.

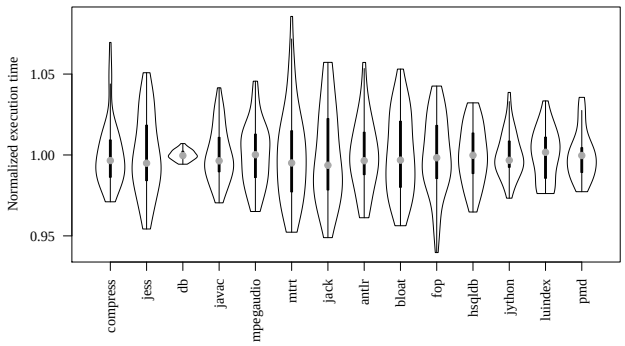

Figure 2. Run-time variability normalized to the mean execution time for start-up performance. These experiments assume 30 VM invocations on the AMD Athlon platform with the GenMS collector and a per-benchmark heap size that is twice as large as the minimal heap size reported in Table 3. The dot represents the median.

6.1 Run-time variability

The basic motivation for this work is that running a Java program introduces run-time variability caused by non-determinism. Figure 2 demonstrates this run-time variability for start-up performance, and Figure 3 shows the same for

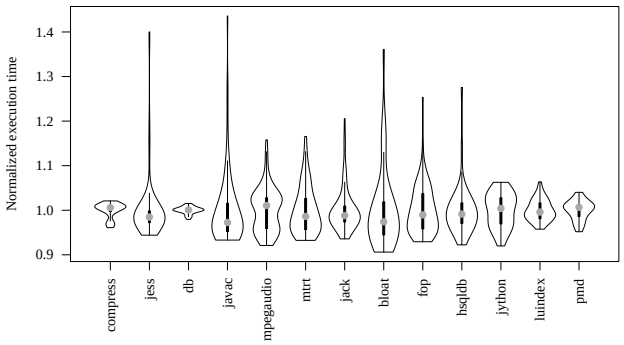

Figure 3. Run-time variability normalized to the mean execution time for steady-state performance. These experiments assume 10 VM invocations and 30 benchmark iterations per VM invocation on the AMD Athlon platform with the GenMS collector and a per-benchmark heap size that is twice as large as the minimal heap size reported in Table 3. The dot represents the median.

steady-state performance[3]. This experiment assumes 30 VM invocations (and a single benchmark iteration) for start-up performance and 10 VM invocations and 30 benchmark iterations per VM invocation for steady-state performance. These graphs show violin plots [13][4]; all values are normalized to a mean of one. A violin plot is very similar to a box plot. The middle point shows the median; the thick vertical line represents the first and third quartile (50% of all the data points are between the first and third quartile); the thin vertical line represents the upper and lower adjacent values (representing an extreme data point or 1.5 times the interquartile range from the median, whichever comes first); and the top and bottom values show the maximum and minimum values. The important difference with a box plot is that the shape of the violin plot represents the density: the wider the violin plot, the higher the density. In other words, a violin plot visualizes a distribution's probability density function whereas a box plot does not.

There are a couple of interesting observations to be made from these graphs. First, run-time variability can be fairly significant, for both startup and steady-state performance. For most of the benchmarks, the coefficient of variation (CoV), defined as the standard deviation divided by the mean, is around 2% and is higher for several benchmarks. Second, the maximum performance difference between the maximum and minimum performance number varies across the benchmarks, and is generally around 8% for startup and 20% for steady-state. Third, most of the violin plots in Figures 2 and 3 show that the measurement data is approximately Gaussian distributed with the bulk weight around the mean. Statistical analyses, such as the Kolmogorov-Smirnov test, do not reject the hypothesis that in most of our measurements the data is approximately Gaussian distributed — in fact, this is the case for more than 90% of the experiments done in this paper involving multiple hardware platforms, benchmarks, GC strategies and heap sizes. However, in a minority of the measurements we observe non-Gaussian distributed data; some of these have skewed distributions or bimodal distributions. Note however that a non-Gaussian distribution does not affect the generality of the proposed statistically rigorous data analysis technique. As discussed in section 3.2.3, the central limit theory does not assume the measurement data to be Gaussian distributed; also, ANOVA is robust towards non-normality.

[3] The `antlr` benchmark does not appear in Figure 3 because we were unable to run more than a few iterations within a single VM invocation.

[4] The graphs shown were made using R – a freely available statistical framework – using the vioplot package available from the CRAN (at `http://www.r-project.org`).

6.2 Evaluating prevalent methodologies

We now compare the prevalent data analysis methodologies against the statistically rigorous data analysis approach advocated in this paper. For doing so, we set up an experiment in which we (pairwise) compare the overall performance of various garbage collectors over a range of heap sizes. We consider the various GC strategies as outlined in section 5, a range of heap sizes from the minimum heap size up to six times this minimum heap size in 0.25 minimum heap size increments — there are 21 heap sizes in total. Computing confidence intervals for the statistically rigorous methodology is done, following section 3, by applying an ANOVA and a Tukey HSD test to compute simultaneous 95% confidence intervals for all the GC strategies per benchmark and per heap size.

To evaluate the accuracy of the prevalent performance evaluation methodologies we consider all possible pairwise GC strategy comparisons for all heap sizes considered. For each heap size, we then determine whether prevalent data analysis leads to the same conclusion as statistically rigorous data analysis. In other words, there are $C_5^2 = 10$ pairwise GC comparisons per heap size and per benchmark. Or, 210 GC comparisons in total across all heap sizes per benchmark.

We now classify all of these comparisons in six categories, see Table 4, and then report the relative frequency of each of these six categories. These results help us better understand the frequency of misleading and incorrect conclusions using prevalent performance methodologies. We make a distinction between overlapping confidence intervals and non-overlapping confidence intervals, according to the statistically rigorous methodology.

Overlapping confidence intervals. Overlapping confidence intervals indicate that the performance differences observed may be due to random fluctuations. As a result, any conclusion taken by a methodology that concludes that one alternative performs better than another is questionable. The only valid conclusion with overlapping confidence intervals

		prevalent methodology	
		performance difference $< \theta$	performance difference $\geq \theta$
statistically rigorous methodology	overlapping intervals	*indicative*	*misleading*
	non-overlapping intervals, same order	*misleading but correct*	*correct*
	non-overlapping intervals, not same order	*misleading and incorrect*	*incorrect*

Table 4. Classifying conclusions by a prevalent methodology in comparison to a statistically rigorous methodology.

is that there is no statistically significant difference between the alternatives.

Performance analysis typically does not state that one alternative is better than another when the performance difference is very small though. To mimic this practice, we introduce a threshold θ to classify decisions: a performance difference smaller than θ is considered a small performance difference and a performance difference larger than θ is considered a large performance difference. We vary the θ threshold from 1% up to 3%.

Now, in case the performance difference by the prevalent methodology is considered large, we conclude the prevalent methodology to be 'misleading'. In other words, the prevalent methodology says there is a significant performance difference whereas the statistics conclude that this performance difference may be due to random fluctuations. If the performance difference is small based on the prevalent methodology, we consider the prevalent methodology to be 'indicative'.

Non-overlapping confidence intervals. Non-overlapping confidence intervals suggest that we can conclude that there are statistically significant performance differences among the alternatives. There are two possibilities for non-overlapping confidence intervals. If the ranking by the statistically rigorous methodology is the same as the ranking by the prevalent methodology, then the prevalent methodology is considered correct. If the methodologies have opposite rankings, then the prevalent methodology is considered to be incorrect.

To incorporate a performance analyst's subjective judgement, modeled through the θ threshold from above, we make one more distinction based on whether the performance difference is considered small or large. In particular, if the prevalent methodology states there is a small difference, the conclusion is classified to be misleading. In fact, there is a statistically significant performance difference, however, the performance difference is small.

We have four classification categories for non-overlapping confidence intervals, see Table 4. If the performance difference by the prevalent methodology is larger than θ, and the ranking by the prevalent methodology equals the ranking by the statistically rigorous methodology, then the prevalent methodology is considered to be 'correct'; if the prevalent methodology has the opposite ranking as the statistically rigorous methodology, the prevalent methodology is considered 'incorrect'. In case of a small performance difference according to the prevalent methodology, and the same ranking as the statistically rigorous methodology, the prevalent methodology is considered to be 'misleading but correct'; in case of an opposite ranking, the prevalent methodology is considered 'misleading and incorrect'.

6.2.1 Start-up performance

We first focus on start-up performance. For now, we limit ourselves to prevalent methodologies that do not use replay compilation. We treat steady-state performance and replay compilation in subsequent sections.

Figure 4 shows the percentage GC comparisons by the prevalent data analysis approaches leading to indicative, misleading and incorrect conclusions for $\theta = 1\%$ and $\theta = 2\%$ thresholds. The various graphs show different hardware platforms and different θ thresholds. The various bars in these graphs show various prevalent methodologies. There are bars for reporting the best, the second best, the worst, the mean and the median performance number; for 3, 5, 10 and 30 VM invocations and a single benchmark iteration — for example, the 'best of 3' means taking the best performance number out of 3 VM invocations. The statistically rigorous methodology that we compare against considers 30 VM invocations and a single benchmark iteration per VM invocation, and considers 95% confidence intervals.

There are a number of interesting observations to be made from these graphs.

- First of all, prevalent methods can be misleading in a substantial fraction of comparisons between alternatives, i.e., the total fraction misleading comparisons ranges up to 16%. In other words, in up to 16% of the comparisons, the prevalent methodology makes too strong a statement saying that one alternative is better than another.
- For a fair number of comparisons, the prevalent methodology can even lead to incorrect conclusions, i.e., the prevalent methodology says one alternative is better (by more than θ percent) than another, whereas the statistically rigorous methodology takes the opposite conclusion based on non-overlapping confidence intervals. For some prevalent methodologies, the fraction of incorrect comparisons can be more than 3%.
- We also observe that some prevalent methodologies perform better than others. In particular, mean and median are consistently better than best, second best and worst. The accuracy of the mean and median methods seems to improve with the number of measurements, whereas the best, second best and worst methods do not.

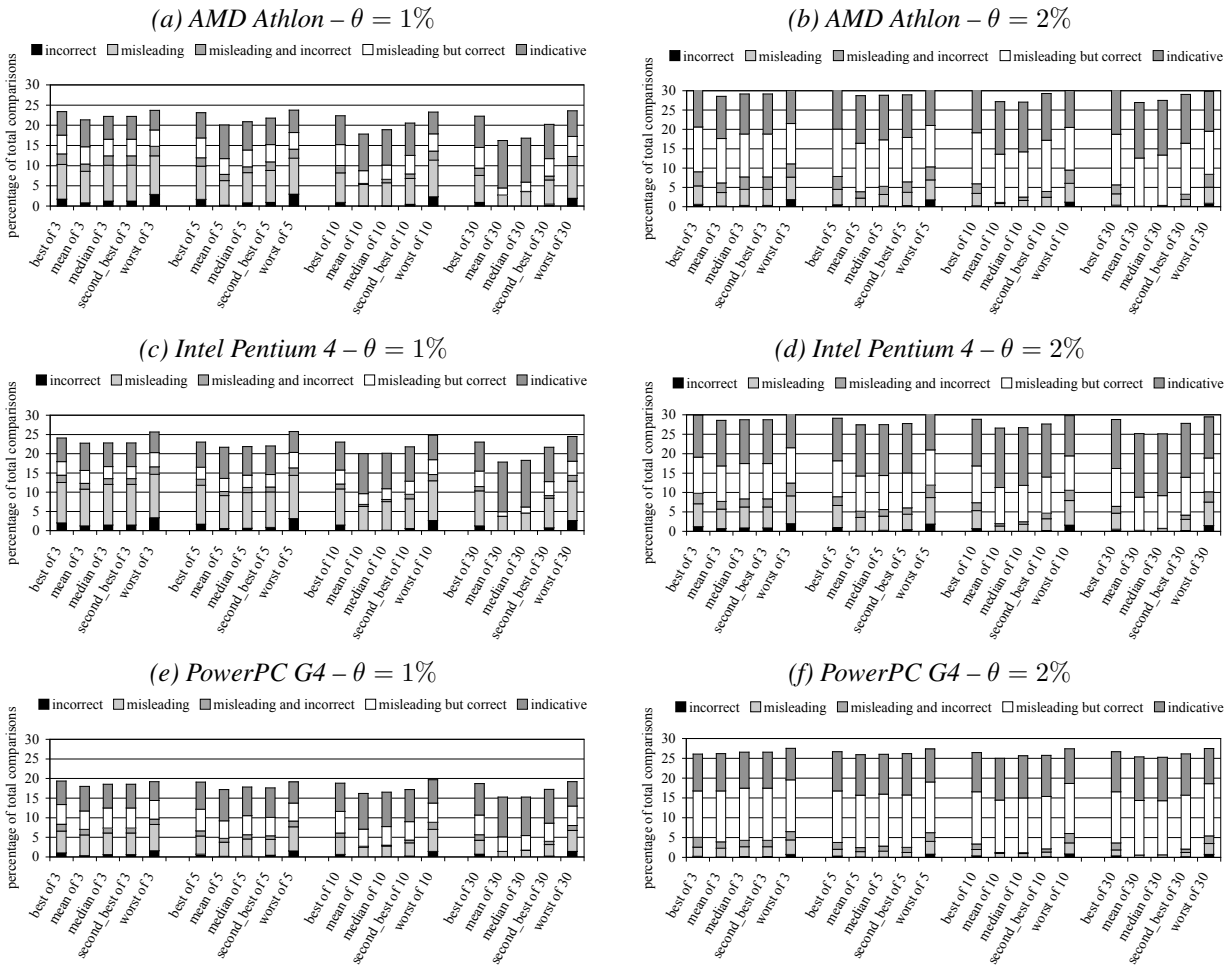

Figure 4. Percentage GC comparisons by prevalent data analysis approaches leading to incorrect, misleading or indicative conclusions. Results are shown for the AMD Athlon machine with $\theta = 1\%$ (a) and $\theta = 2\%$ (b), for the Intel Pentium 4 machine with $\theta = 1\%$ (c) and $\theta = 2\%$ (d), and for the PowerPC G4 with $\theta = 1\%$ (e) and $\theta = 2\%$ (f).

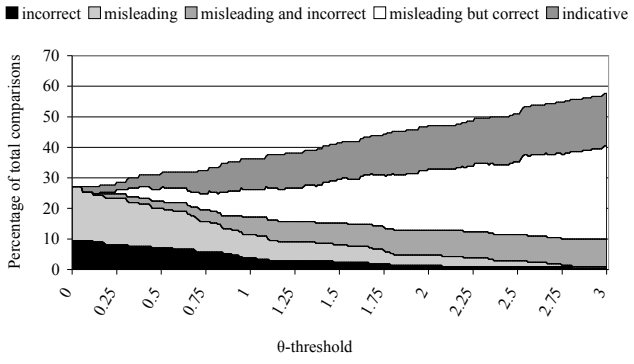

Figure 5. The classification for javac as a function of the threshold $\theta \in [0; 3]$ for the 'best' prevalent method, on the AMD Athlon.

- Increasing the θ threshold reduces the number of incorrect conclusions by the prevalent methodologies and at the same time also reduces the number of misleading and correct conclusions. By consequence, the number of misleading-but-correct, misleading-and-incorrect and indicative conclusions increases, or, in other words, the conclusiveness of a prevalent methodology reduces with an increasing θ threshold. Figure 5 shows the classification as a function of the θ threshold for the javac benchmark, which we found to be a representative example benchmark. The important conclusion here is that increasing the θ threshold for a prevalent methodology does not replace a statistically rigorous methodology.
- One final interesting observation that is consistent with the observations made by Blackburn et al. [7], is that the results presented in Figure 4 vary across different hardware platforms. In addition, the results also vary across benchmarks, see Figure 6 which shows per-benchmark

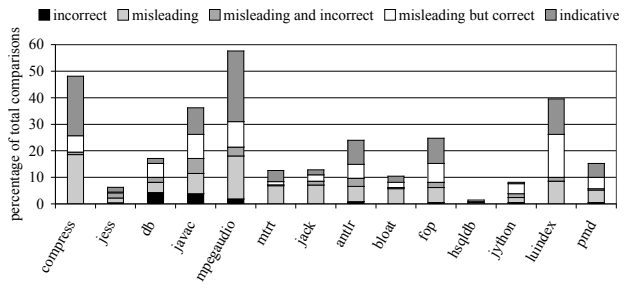

Figure 6. Per-benchmark percentage GC comparisons by the 'best' method classified as misleading, incorrect and indicative on the AMD Athlon machine with $\theta = 1\%$.

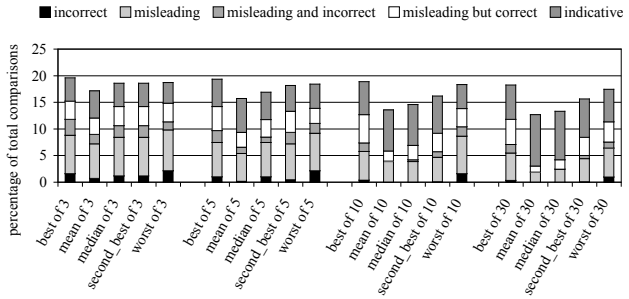

Figure 7. The (in)accuracy of comparing the GenMS GC strategy against four other GC strategies using prevalent methodologies, for $\theta = 1\%$ on the AMD Athlon machine.

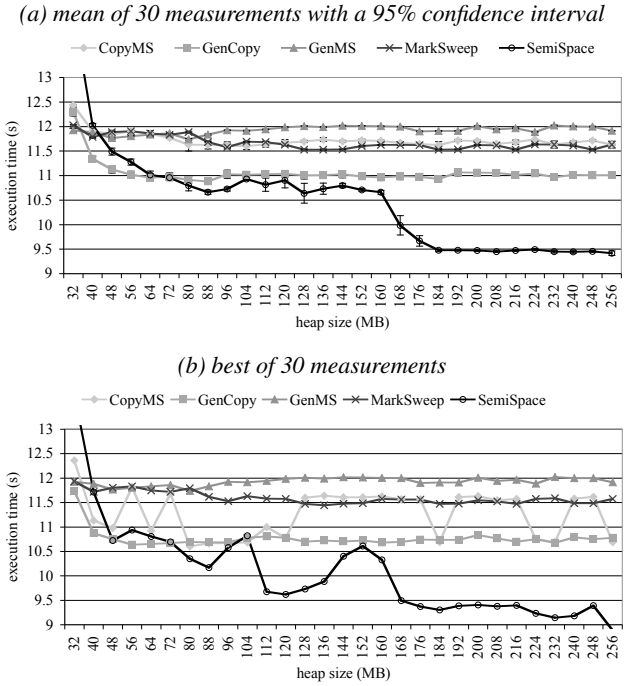

Figure 8. Startup execution time (in seconds) for db as a function of heap size for five garbage collectors; mean of 30 measurements with 95% confidence intervals (top) and best of 30 measurements (bottom).

results for the 'best' prevalent method; we obtained similar results for the other methods. Some benchmarks are more sensitive to the data analysis method than others. For example, jess and hsqldb are almost insensitive, whereas other benchmarks have a large fraction misleading and incorrect conclusions; db and javac for example show more than 3% incorrect conclusions.

A VM developer use case. The evaluation so far quantified comparing *all* GC strategies against all other GC strategies, a special use case. Typically, a researcher or developer is merely interested in comparing a new feature against already existing approaches. To mimic this use case, we compare *one* GC strategy, GenMS, against all other four GC strategies. The results are shown in Figure 7 and are very much in line with the results presented in Figure 4: prevalent data analysis methods are misleading in many cases, and in some cases even incorrect.

An application developer use case. Our next case study takes a look from the perspective of an application developer by looking at the performance of a single benchmark. Figure 8 shows two graphs for db for the best of 30 and the confidence interval based performance evaluation methods. The different curves represent different garbage collectors. These graphs clearly show that different conclusions may be taken depending on the evaluation method used. For example, for heap sizes between 80MB and 120MB, one would conclude using the 'best' method that CopyMS clearly outperforms MarkSweep and performs almost equally well as GenCopy. However, the confidence intervals show that the performance difference between CopyMS and MarkSweep could be due to random fluctuations, and in addition, the statistically rigorous method clearly shows that GenCopy substantially outperforms CopyMS.

Figure 9 shows similar graphs for antlr. Figure 10 suggests that for large heap sizes (6 to 20 times the minimum) most of the performance differences observed between the CopyMS, GenCopy and SemiSpace garbage collectors are due to non-determinism. These experiments clearly illustrate that both experimental design and data analysis are important factors in a Java performance analysis methodology. Experimental design may reveal performance differences among design alternatives, but without statistical data analysis, we do not know if these differences are meaningful.

6.2.2 Steady-state performance

Figure 11 shows normalized execution time (averaged over a number of benchmarks) as a function of the number iterations for a single VM invocation. This graph shows that it takes a number of iterations before steady-state performance is reached: the first 3 iterations obviously seem to be part

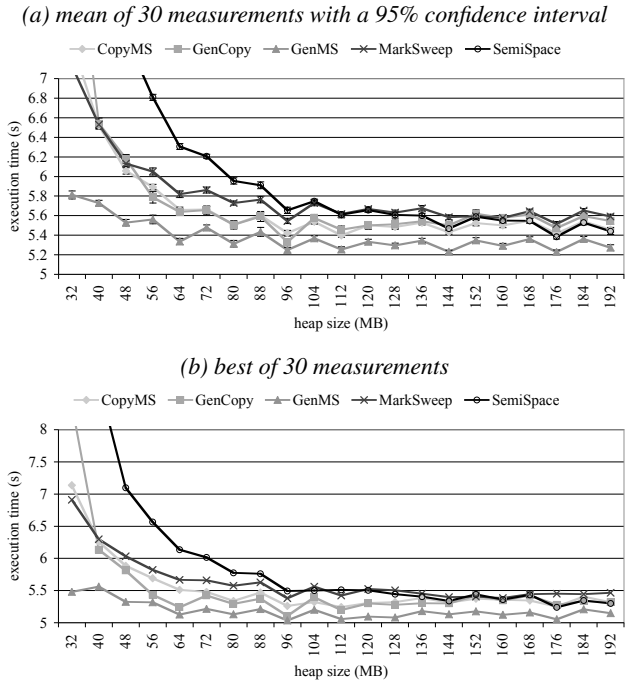

Figure 9. Startup execution time (in seconds) for antlr as a function of heap size for five garbage collectors; mean of 30 measurements with 95% confidence intervals (top) and best of 30 measurements (bottom).

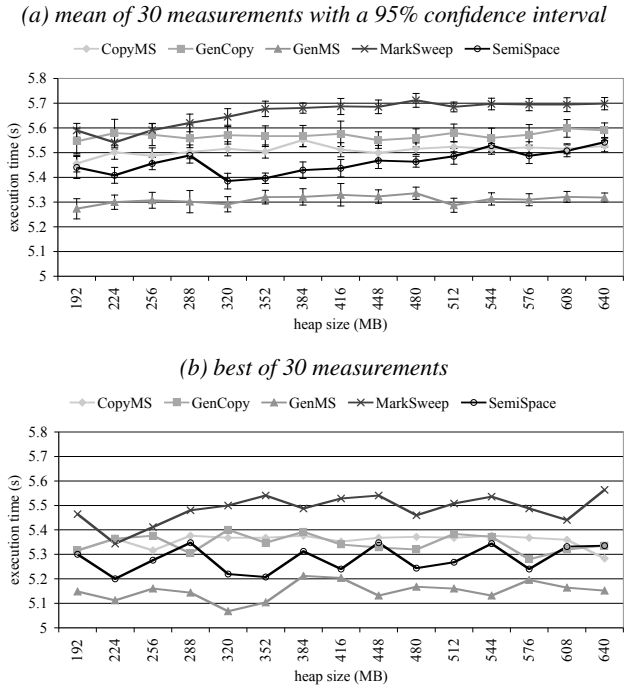

Figure 10. Startup execution time (in seconds) for antlr over a range of *large* heap sizes (6 to 20 times the minimum heap size) for five garbage collectors; mean of 30 measurements with 95% confidence intervals (top) and best of 30 measurements (bottom).

of startup performance, and it takes more than 10 iterations before we actually reach steady-state performance.

For quantifying steady-state performance, following Section 4.2, we retain $k = 10$ iterations per VM invocation for which the CoV is smaller than 0.02. Figure 12 compares three prevalent steady-state performance methodologies against the statistically rigorous approach: (i) best of median (take the median per iteration across all VM invocations, and then select the best median iteration), (ii) best performance number, and (iii) second best performance number across all iterations and across all VM invocations. For these prevalent methods we consider 1, 3 and 5 VM invocations and 3, 5, 10 and 30 iterations per VM invocation. The general conclusion concerning the accuracy of the prevalent methods is similar to those for startup performance. Prevalent methods are misleading in more 20% of the cases for a $\theta = 1\%$ threshold, more than 10% for a $\theta = 2\%$ threshold, and more than 5% for a $\theta = 3\%$ threshold. Also, the number of incorrect conclusions is not negligible (a few percent for small θ thresholds).

6.2.3 Replay compilation

Replay compilation is an increasingly popular experimental design setup that removes the non-determinism from compilation in the VM. It is particularly convenient for specific topics of research. One such example is GC research: replay compilation enables the experimenter to focus on GC performance while controlling non-determinism by the VM's adaptive compilation and optimization subsystem.

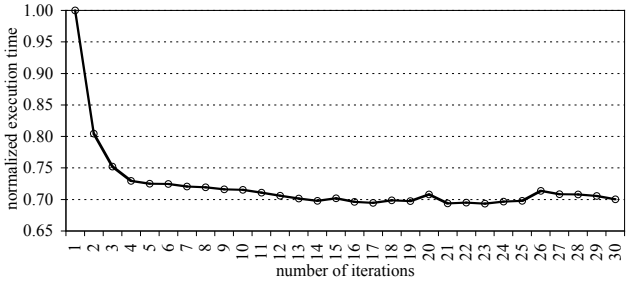

Figure 11. Normalized execution time as a function of the number of iterations on the AMD Athlon machine.

The goal of this section is twofold. First, we focus on experimental design and quantify how replay compilation compares against non-controlled compilation, assuming statistically rigorous data analysis. Second, we compare prevalent data analysis techniques against statistically rigorous data analysis under replay compilation.

In our replay compilation approach, we analyze 7 benchmark runs in separate VM invocations and take the optimal (yielding the shortest execution time) compilation plan. We also evaluated the majority plan and obtained similar results.

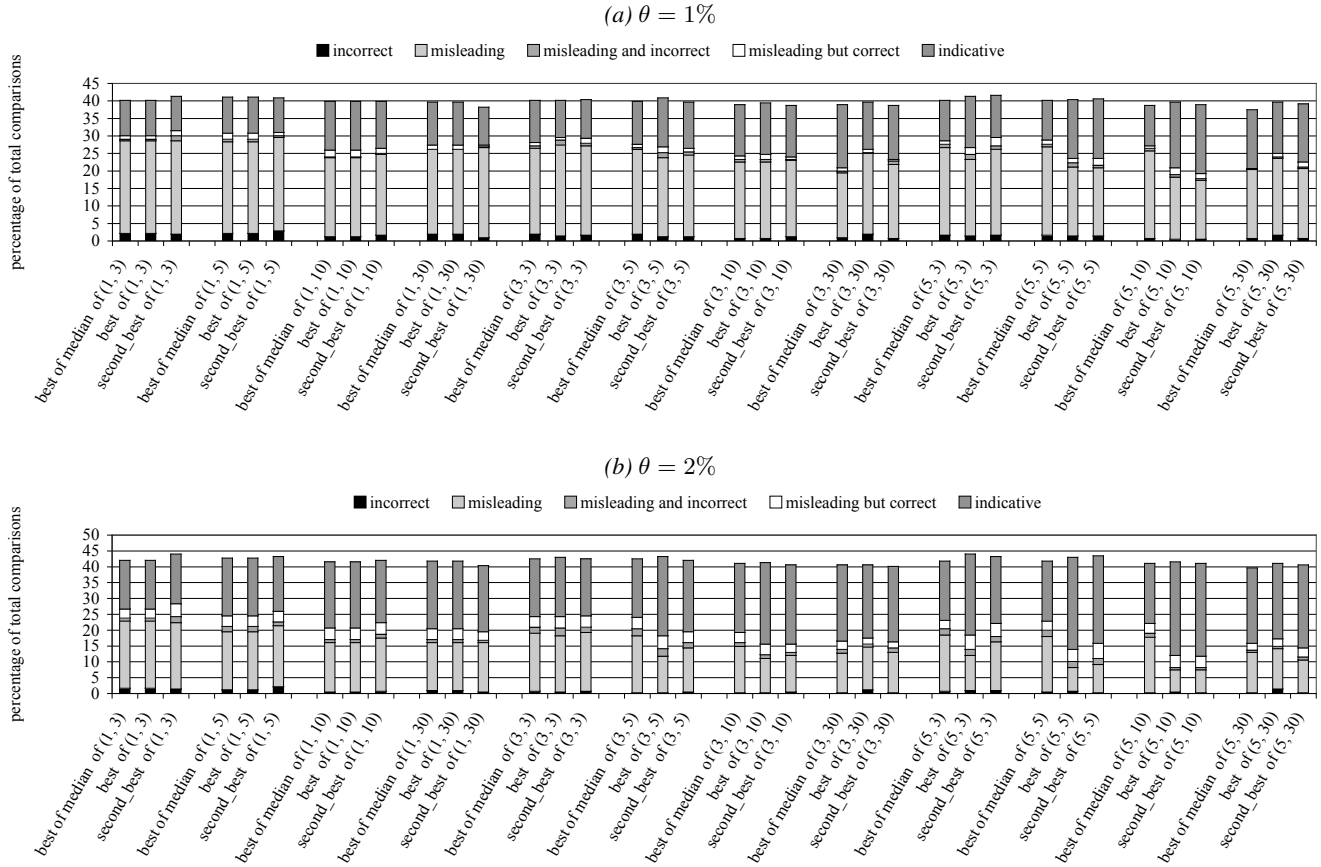

Figure 12. The (in)accuracy of prevalent methodologies compared to steady-state performance: (x, y) denotes x VM invocations and y iterations per VM invocation; for SPECjvm98 on the AMD Athlon machine.

The compilation plan is derived for start-up performance using the GenMS configuration with a 512MB heap size. The timing run consists of two benchmark iterations: the first one, called mix, includes compilation activity, and the second one, called stable, does not include compilation activity. A full GC is performed between these two iterations. The timing runs are repeated multiple times (3, 5, 10 and 30 times in our setup).

Experimental design. Figure 13 compares mix replay versus startup performance as well as stable replay versus steady-state performance, assuming non-controlled compilation. We assume statistically rigorous data analysis for both the replay compilation and non-controlled compilation experimental setups. We classify all GC comparisons in three categories: 'agree', 'disagree' and 'inconclusive', see Table 5, and display the 'disagree' and 'inconclusive' categories in Figure 13. We observe replay compilation and non-controlled compilation agree in 56% to 72% of all cases, and are inconclusive in 17% (DaCapo mix versus startup) to 37% (SPECjvm98 stable versus steady-state) of all cases. In up to 12% of all cases, see SPECjvm98 mix versus startup

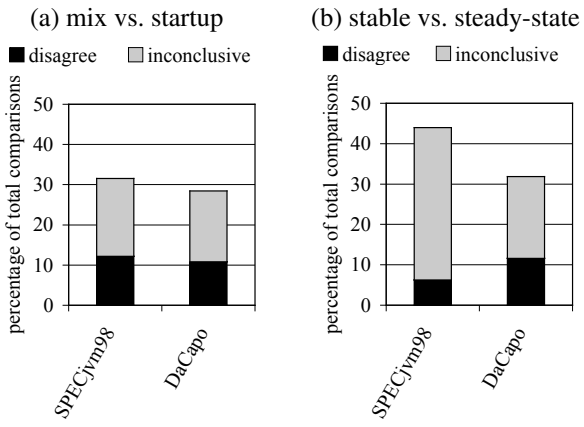

Figure 13. Comparing mix replay compilation versus startup performance (a), and stable replay compilation versus steady-state performance (b) under non-controlled compilation using statistically rigorous data analysis on the AMD Athlon platform.

		replay compilation		
		overlapping intervals	non-overlapping intervals	
			$A > B$	$B > A$
non-controlled compilation	overlapping intervals	*agree*	*inconclusive*	
	non-overlapping intervals, $A > B$	*inconclusive*	*agree*	*disagree*
	non-overlapping intervals, $B > A$	*inconclusive*	*disagree*	*agree*

Table 5. Classifying conclusions by replay compilation versus non-controlled compilation.

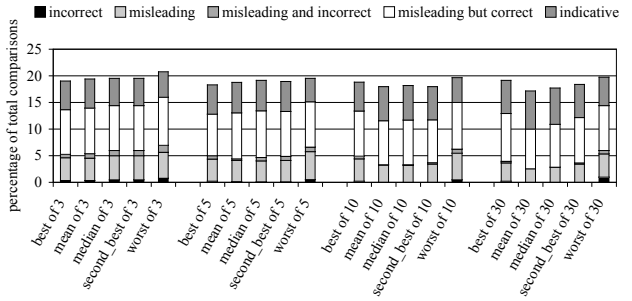

Figure 14. Comparing prevalent data analysis versus statistically rigorous data analysis under mix replay compilation, assuming $\theta = 1\%$ on the AMD Athlon platform.

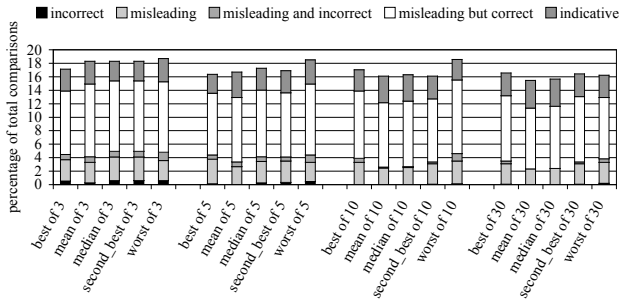

Figure 15. Comparing prevalent data analysis versus statistically rigorous data analysis under stable replay compilation, assuming $\theta = 1\%$ on the AMD Athlon platform.

and DaCapo stable versus steady-state, both experimental designs disagree. These two experimental designs offer different garbage collection loads and thus expose different space-time trade-offs that the collectors make.

Data analysis. We now assume replay compilation as the experimental design setup, and compare prevalent data analysis versus statistically rigorous data analysis. Figures 14 and 15 show the results for mix replay versus startup performance, and stable replay versus steady-state performance, respectively. These results show that prevalent data analysis can be misleading under replay compilation for startup performance: the fraction misleading conclusions is around 5%, see Figure 14. For steady-state performance, the number of misleading conclusions is less than 4%, see Figure 15.

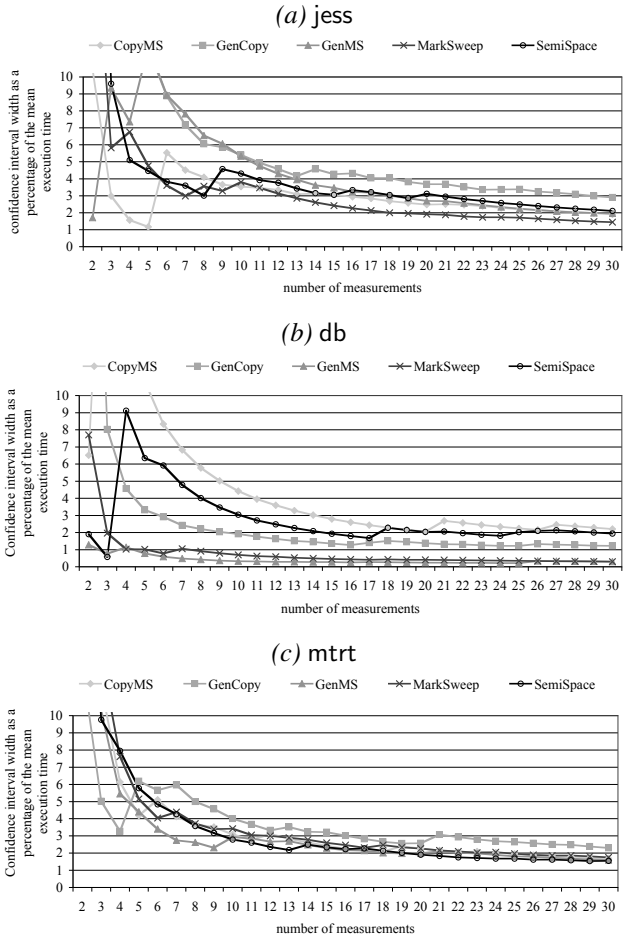

Figure 16. Confidence width as a percentage of the mean (on the vertical axis) as a function of the number of measurements taken (on the horizontal axis) for three benchmarks: jess (top), db (middle) and mtrt (bottom).

6.3 Statistically rigorous performance evaluation in practice

As discussed in Section 3, the width of the confidence interval is a function of the number of measurements n. In general, the width of the confidence interval decreases with an increasing number of measurements as shown in Figure 16. The width of the 95% confidence interval is shown as a percentage of the mean (on the vertical axis) and as a function of the number of measurements taken (on the horizontal axis). We show three example benchmarks: jess, db and mtrt for a

80MB heap size on the AMD Athlon machine. The various curves represent different garbage collectors for start-up performance. The interesting observation here is that the width of the confidence interval largely depends on both the benchmark and the garbage collector. For example, the width of the confidence interval for the GenCopy collector for jess is fairly large, more than 3%, even for 30 measurements. For the MarkSweep and GenMS collectors for db on the other hand, the confidence interval is much smaller, around 1% even after less than 10 measurements.

These observations motivated us to come up with an automated way of determining how many measurements are needed to achieve a desired confidence interval width. For example, for db and the MarkSweep and GenMS collectors, a handful of measurements will suffice to achieve a very small confidence interval, whereas for jess and the GenCopy collector many more measurements are needed. JavaStats consists of a script (to initiate multiple VM invocations) and a harness (to initiate multiple benchmark iterations within a single VM invocation) that computes the width of the confidence interval while the measurements are being taken. It takes as input the desired confidence interval width (for example 2% or 3%) for a given confidence level and a maximum number of VM invocations and benchmark iterations. JavaStats stops the measurements and reports the confidence interval as soon as the desired confidence interval width is achieved or the maximum number of VM invocations and benchmark iterations is reached.

Figure 17 reports the number of VM invocations required for start-up performance to achieve a 2% confidence interval width with a maximum number of VM invocations, $p = 30$ for jess, db and mtrt on the AMD Athlon as a function of heap size for the five garbage collectors. The interesting observation here is that the number of measurements taken varies from benchmark to benchmark, from collector to collector and from heap size to heap size. This once again shows why an automated way of collecting measurements is desirable. Having to take fewer measurements for a desired level of confidence speeds up the experiments compared to taking a fixed number of measurements.

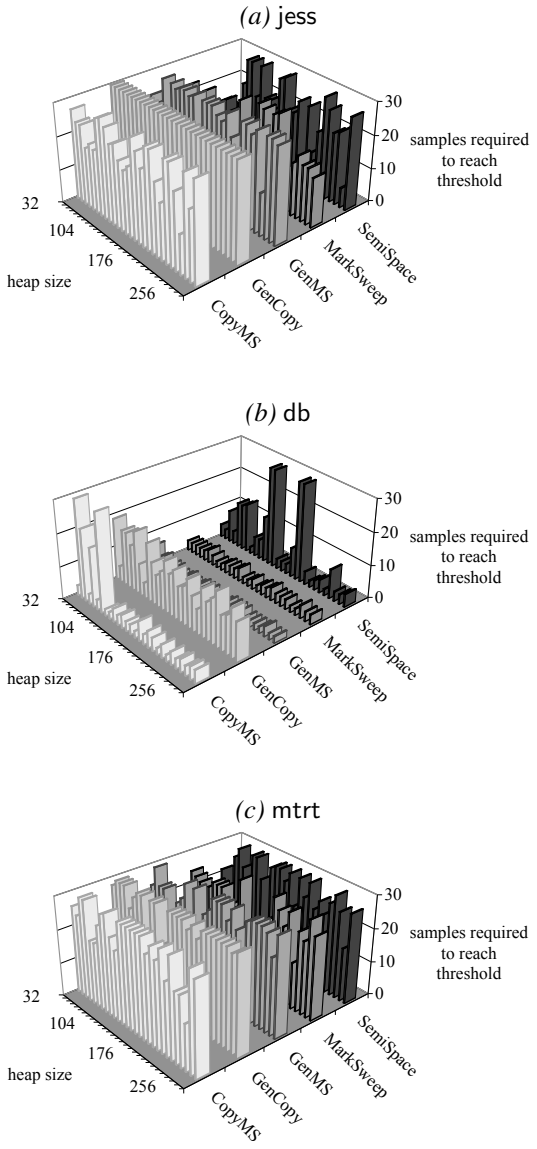

Figure 17. Figure shows how many measurements are required before reaching a 2% confidence interval on the AMD Athlon machine.

7. Summary

Non-determinism due to JIT compilation, thread scheduling, garbage collection and various system effects, makes quantifying Java performance far from being straightforward. Prevalent data analysis approaches deal with non-determinism in a wide variety of ways. This paper showed that prevalent data analysis approaches can be misleading and can even lead to incorrect conclusions.

This paper introduced statistically rigorous Java performance methodologies for quantifying Java startup and steady-state performance. In addition, it presented JavaStats, publicly available software to automatically perform rigorous performance evaluation. For startup performance, we run multiple VM invocations executing a single benchmark iteration and subsequently compute confidence intervals. For steady-state performance, we run multiple VM invocations, each executing multiple benchmark iterations. We then compute a confidence interval based on the benchmark iterations across the various VM invocations once performance variability drops below a given threshold.

We believe this paper is a step towards statistical rigor in various performance evaluation studies. Java performance analysis papers, and papers presenting experimental results in general, very often report performance improvements between two or more alternatives. Most likely, if the perfor-

mance differences between the alternatives are large, a statistically rigorous method will not alter the overall picture nor affect the general conclusions obtained using prevalent methods. However, for relatively small performance differences (that are within the margin of experimental error), not using statistical rigor may lead to incorrect conclusions.

Acknowledgments

We would like to thank Steve Blackburn, Michael Hind, Matthew Arnold, Kathryn McKinley, and the anonymous reviewers for their valuable comments — their detailed suggestions greatly helped us improving this paper. Andy Georges is supported by Ghent University. Dries Buytaert is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research–Flanders (Belgium) (FWO–Vlaanderen).

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, Oct. 2000.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *OOPSLA*, pages 111–129, Nov. 2002.

[3] K. Barabash, Y. Ossia, and E. Petrank. Mostly concurrent garbage collection revisited. In *OOPSLA*, pages 255–268, Nov. 2003.

[4] O. Ben-Yitzhak, I. Goft, E. K. Kolodner, K. Kuiper, and V. Leikehman. An algorithm for parallel incremental compaction. In *ISMM*, pages 207–212, Feb. 2003.

[5] S. Blackburn, P. Cheng, and K. McKinley. Myths and reality: The performance impact of garbage collection. In *SIGMETRICS*, pages 25–36, June 2004.

[6] S. Blackburn, P. Cheng, and K. McKinley. Oil and water? High performance garbage collection in Java with JMTk. In *ICSE*, pages 137–146, May 2004.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[8] S. M. Blackburn and K. S. McKinley. In or out?: Putting write barriers in their place. In *ISMM*, pages 281–290, June 2002.

[9] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, pages 344–358, Oct. 2003.

[10] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *OOPSLA*, pages 169–186, Oct. 2003.

[11] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE*, pages 111–121, June 2006.

[12] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-priented applications. In *OOPSLA*, pages 251–269, Oct. 2004.

[13] J.L. Hintze, and R.D. Nelson. Violin Plots: A Box Plot-Density Trace Synergism In *The American Statistician*, Volume 52(2), pages 181–184, May 1998.

[14] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA*, pages 69–80, Oct. 2004.

[15] R.A. Johnson and D.W. Wichern Applied Multivariate Statistical Analysis Prentice Hall, 2002

[16] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[17] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. Javana: A system for building customized Java program analysis tools. In *OOPSLA*, pages 153–168, Oct. 2006.

[18] P. McGachey and A. L. Hosking. Reducing generational copy reserve overhead with fallback compaction. In *ISMM*, pages 17–28, June 2006.

[19] J. Neter, M. H. Kutner, W. Wasserman, and C. J. Nachtsheim Applied Linear Statistical Models WCB/McGraw-Hill, 1996.

[20] N. Sachindran and J. E. B. Moss. Mark-copy: Fast copying GC with less space overhead. In *OOPSLA*, pages 326–343, Oct. 2003.

[21] K. Sagonas and J. Wilhelmsson. Mark and split. In *ISMM*, pages 29–39, June 2006.

[22] D. Siegwart and M. Hirzel. Improving locality with parallel hierarchical copying GC. In *ISMM*, pages 52–63, June 2006.

[23] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[24] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM*, pages 57–72, May 2004.

[25] C. Zhang, K. Kelsey, X. Shen, C. Ding, M. Hertz, and M. Ogihara. Program-level adaptive memory management. In *ISMM*, pages 174–183, June 2006.